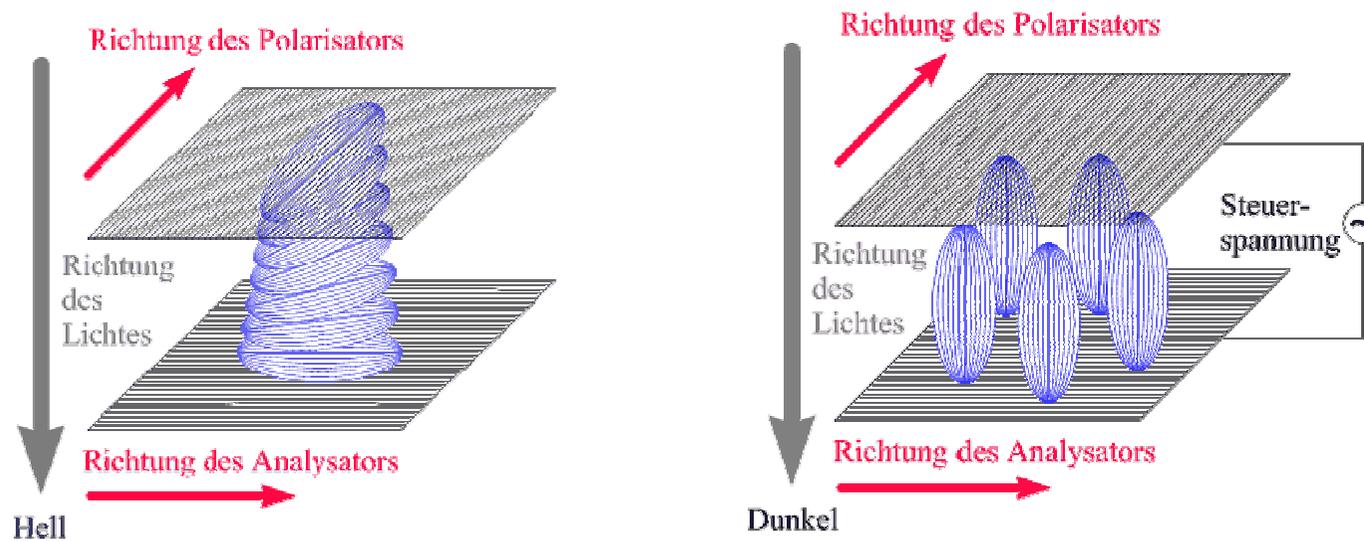


Übersicht

1. Anzeigegeräte
2. Framebuffer
3. Grundlagen 3D Computergrafik
4. Polygongrafik, Z-Buffer
5. Texture-Mapping/Shading
6. GPU
7. Programmierbare Shader

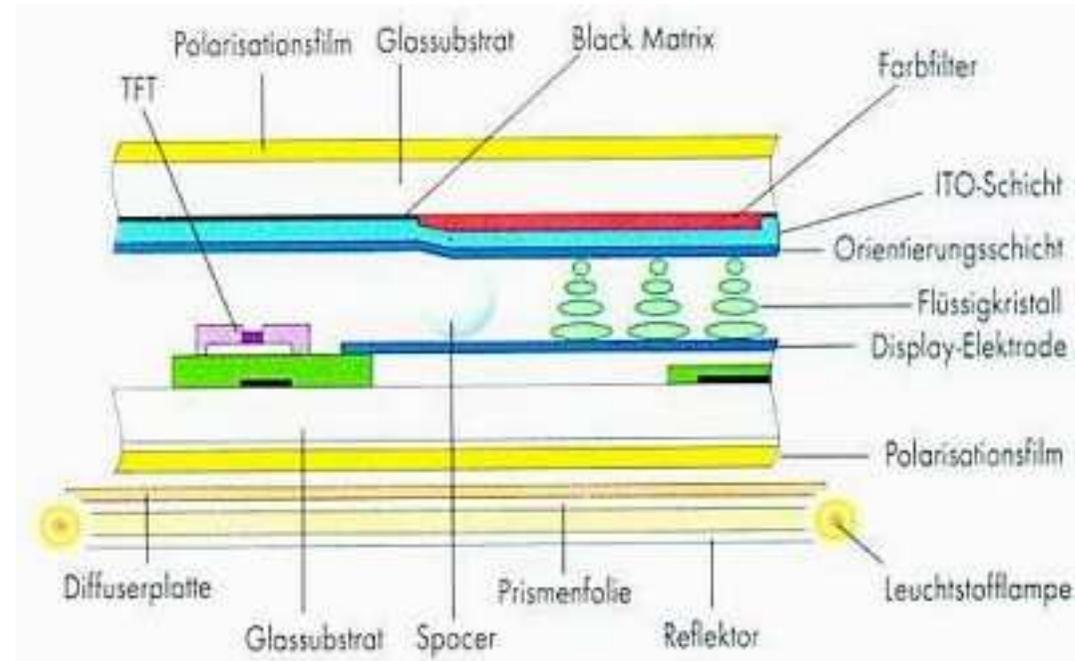
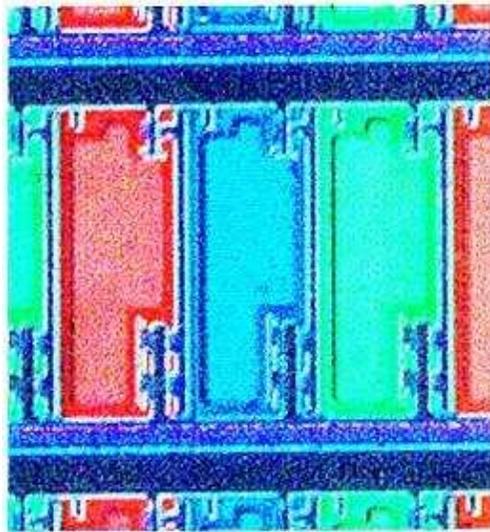
LCD/TFT Technik

- Rotation der Licht-Polarisationsebene
- Bei Anlegen eines elektr. Feldes keine Rotation
- Intensitätssteuerung durch Feldstärke

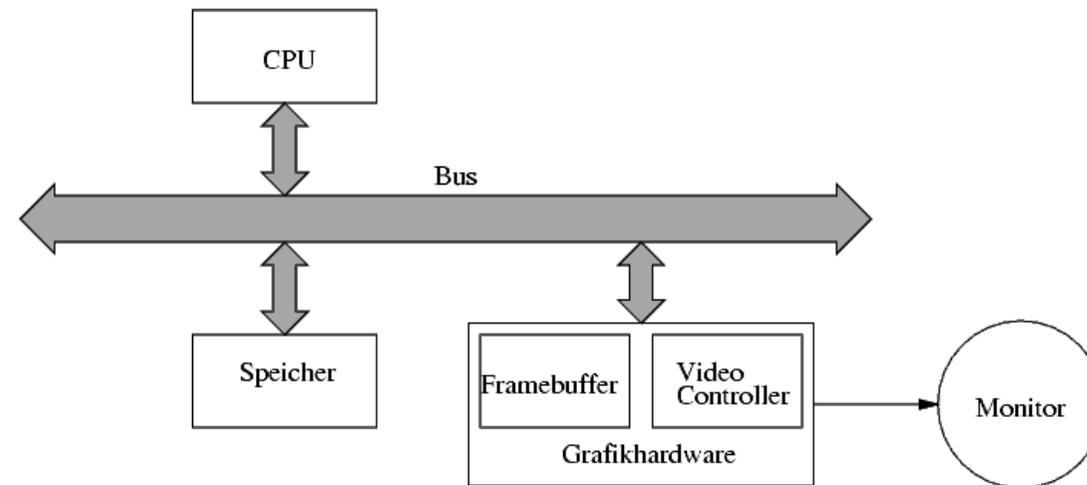


LCD/TFT Technik

- Pro Pixel/Farbkomponente ein Transistor
- Schnelle Ansteuerung individueller Punkte
- Richtungsabhängigkeit

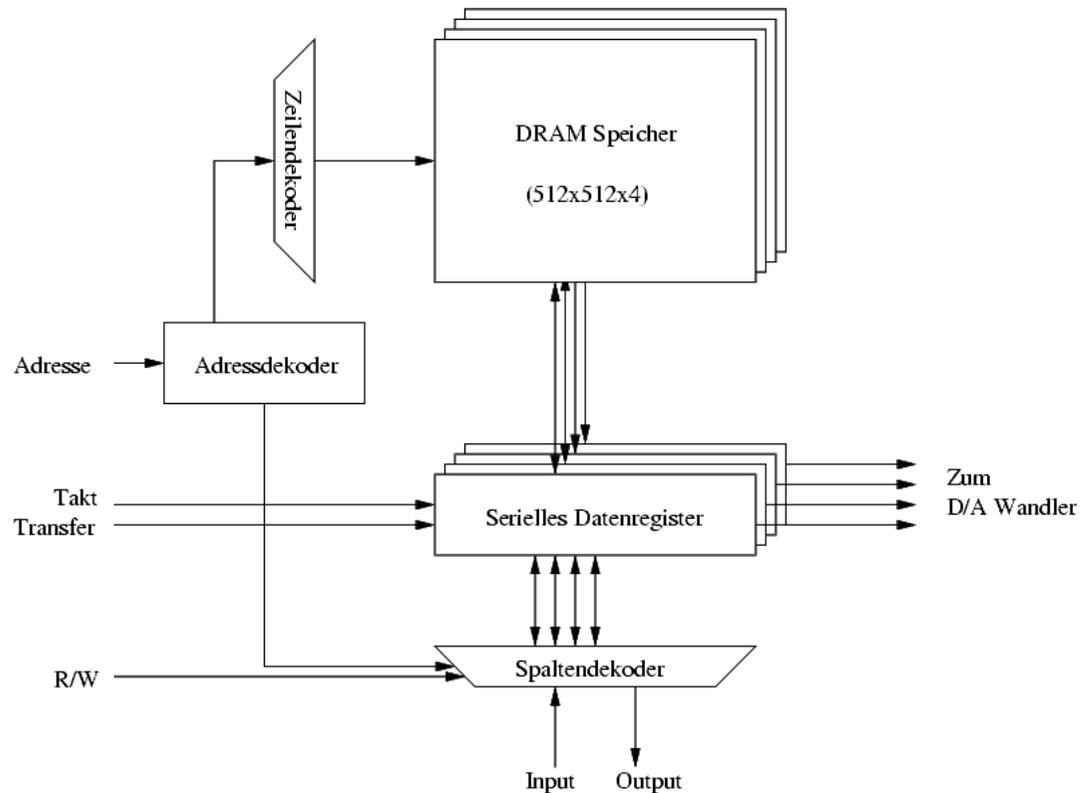


Framebuffer



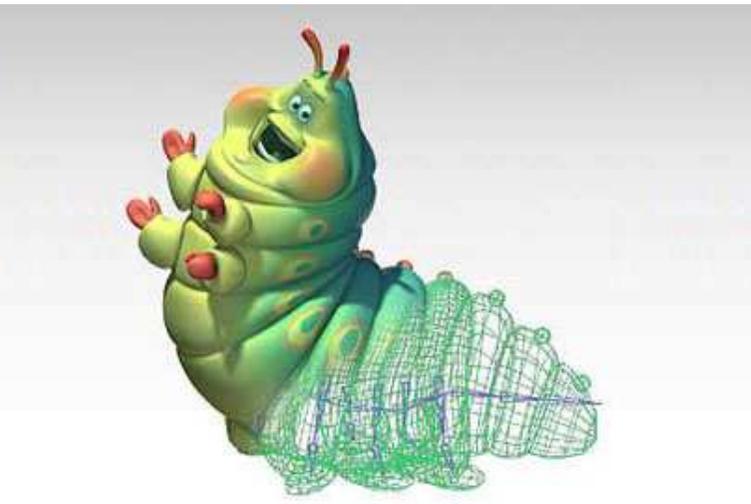
- Gleichzeitiges Lesen und Schreiben notwendig
- Dual-ported Memory

Framebuffer

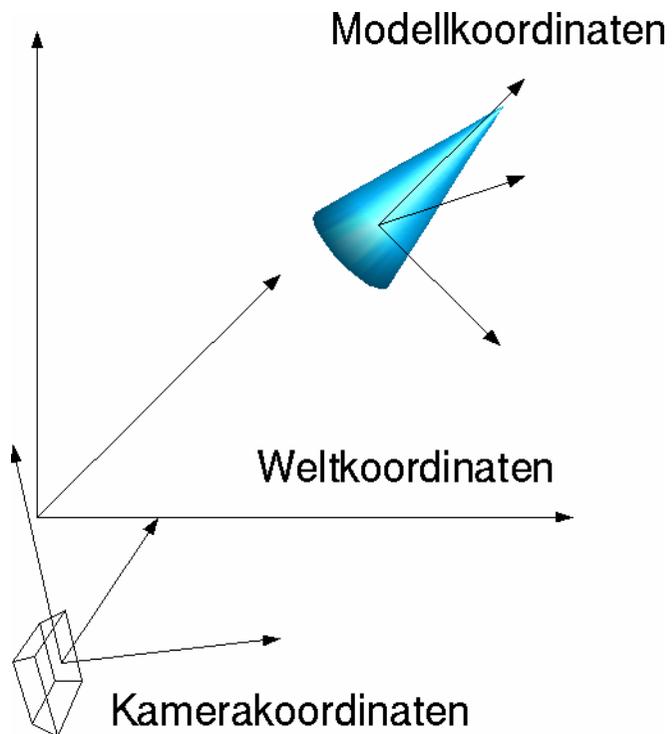


3D Computergrafik

- 3D Transformationen
- 3D/2D Projektion
- Schattierung
- Texturen
- Zeichnen von Polygonen mit Tiefeninformation



3D Transformation



Unterscheidung zwischen:

- Weltkoordinatensystem
- Kamerakoordinatensystem
- Modellkoordinatensystem

3D Transformation

- Benutzung von *homogenen* Koordinaten
(vgl. Projektive Geometrie)

⇒ Ergänzung um homogene Komponente

Bspw.: Vektor $\mathbf{p} \in \mathcal{K}^3$ ⇒ $\begin{bmatrix} p_x \\ p_y \\ p_z \\ p_w \end{bmatrix}$

- Kompakte Darstellung von Translationen/Rotationen
- Projektion durch homogene Division

3D Transformation

Beispiel Koordinatentransformation:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & t_x \\ m_{21} & m_{22} & m_{23} & t_y \\ m_{31} & m_{32} & m_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Rotation/Skalierung
 Translation

Beispiel Perspektivische Projektion:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad \left[\begin{array}{cccc} x & y & z & 1 \\ w & w & w & 1 \end{array} \right] \Rightarrow [x_s \quad y_s \quad z_s]$$

(homogene Division)

3D Beschleunigung – Antreibende Faktoren

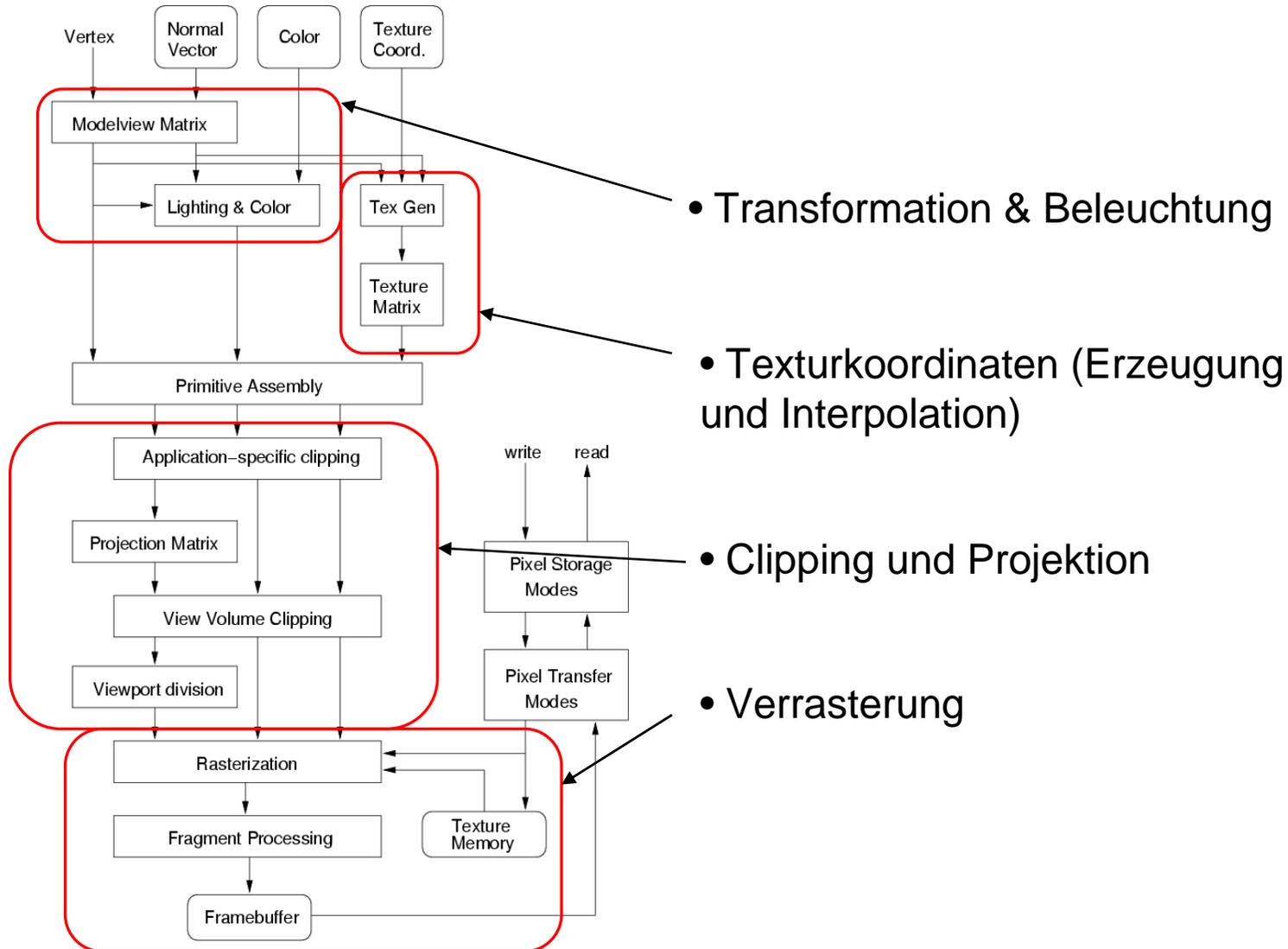
- Gestiegene Anforderungen im CAD Bereich
- Simulatoren (speziell mil. Anwendungen)
- Visualisierung großer Datenmengen

3D Beschleunigung

Computerspiele



3D Grafikpipeline (OpenGL)



• Transformation & Beleuchtung

• Texturkoordinaten (Erzeugung und Interpolation)

• Clipping und Projektion

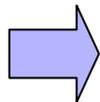
• Verrasterung

Zeichnen von Polygonen

- 1. Generation der 3D-Beschleunigung
- CPU stellt transformierte Daten (ggf. Texturen, Farben)
- Zeichnen von Polygonen durch Grafikhardware
- Interpolation von Farbe, Texturkoordinaten, Z-Koordinaten

Problematisch:

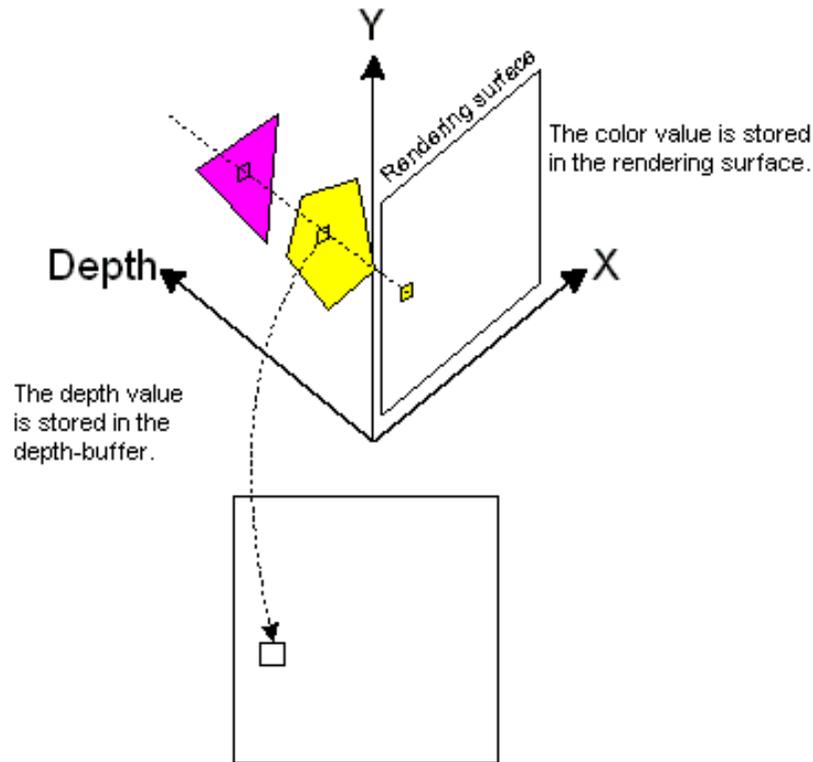
- Verdeckungen aufgrund der 3D Daten



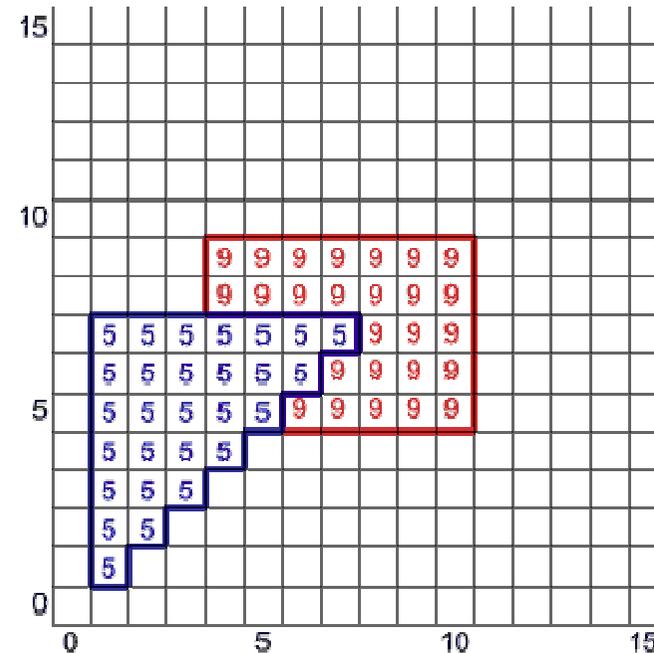
Z-Buffer



Z-Buffer



Depth-buffer (same dimensions as the rendering surface)



- Speicher auf Grafikkarte (üblicherweise im Framebuffer)
- Speichert n -Bit Tiefeninformationen pro Pixel
- Hardware führt Vergleiche während des Zeichnens aus

Texture-Mapping



- Hinzufügen von Detailinformationen auf Polygone
- Detailinformationen sind i.A. 2D-Bilddaten
- Erhöhung des Fotorealismus
- Rechenaufwendig

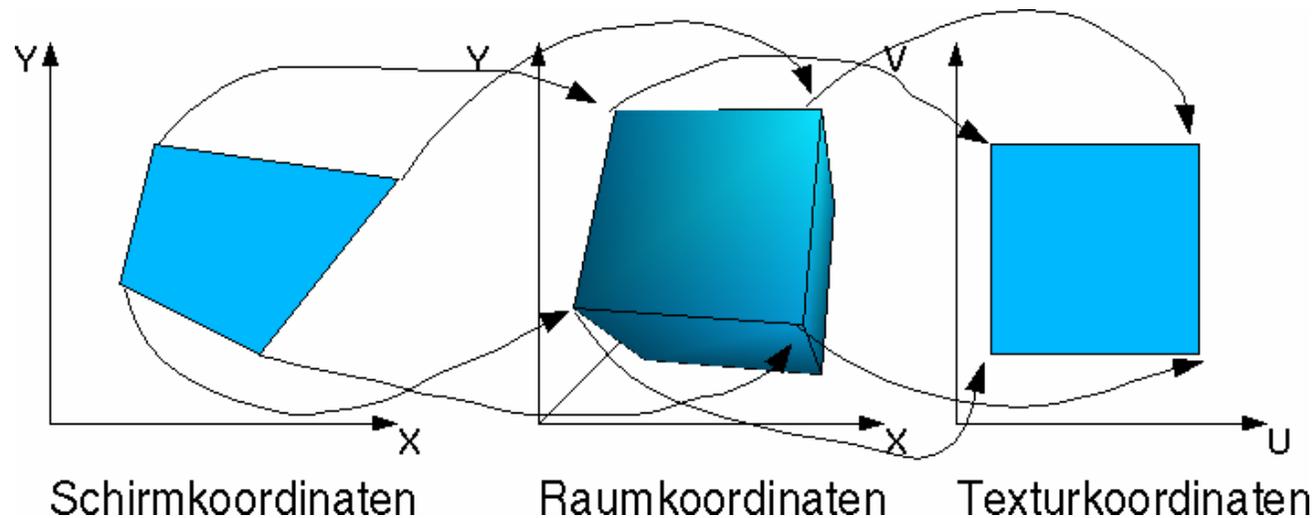
Eingabevektor für die Grafikkarte:

$X/Y/Z/W$ Koordinatenvektoren, Texturkoordinaten (u,v) , Textur-ID

➡ Texturspeicher / ALU auf der Grafikkarte

Texture-Mapping

- Üblicherweise: Mapping Schirmkoordinaten \rightarrow Raumkoordinaten



Problem:

- Rückprojektion für jeden Bildschirmpixel

\rightarrow *Rational-Lineare Interpolation*

Texturen – RL Interpolation

- Projektion/Homogene Division an den Eckpunkten

- Lineare Interpolation von $\begin{bmatrix} x & y & z & 1 & u & v \\ w & w & w & w & w & w \end{bmatrix}$

- Für jeden Pixel:

$$u = \frac{u/w}{1/w} \quad v = \frac{v/w}{1/w}$$

➡ Gute Realisierbarkeit in Hardware

Shading

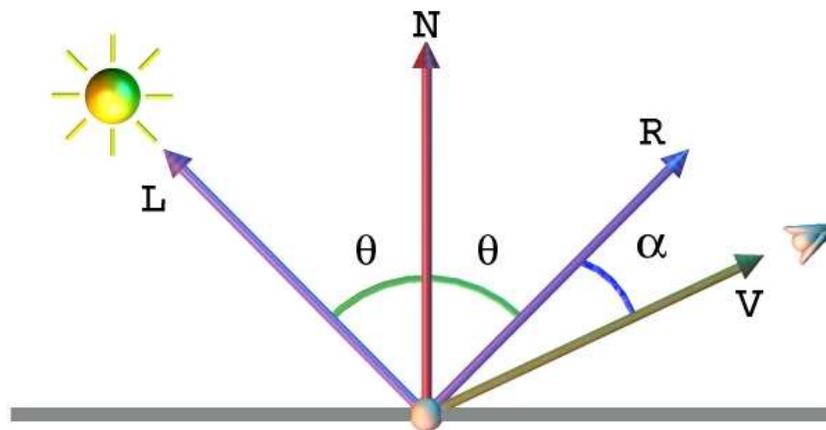
- Einbeziehen von Lichtquellen
- Erhöhung des Fotorealismus
- Üblicherweise lokale Beleuchtungsverfahren
 - Flat Shading
 - Gouraud Shading
 - Phong Shading



Shading

$$I = I_a k_a + f_{att} I_l (k_d (\mathbf{N}^T \mathbf{L}) + k_s (\mathbf{R}^T \mathbf{V})^n)$$

(Phong Beleuchtungsmodell)

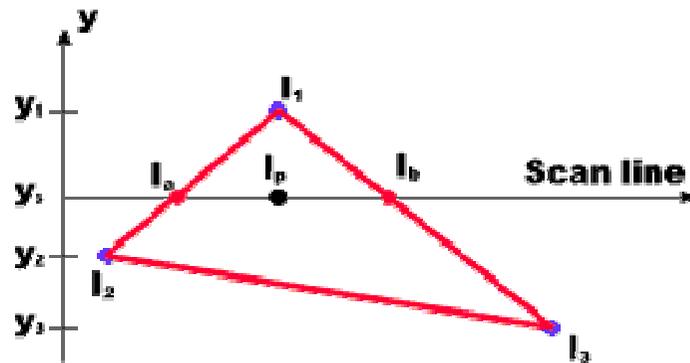


Benötigte Arithmetik:

Skalarprodukt, Multiplikation, Addition, Potenzierung

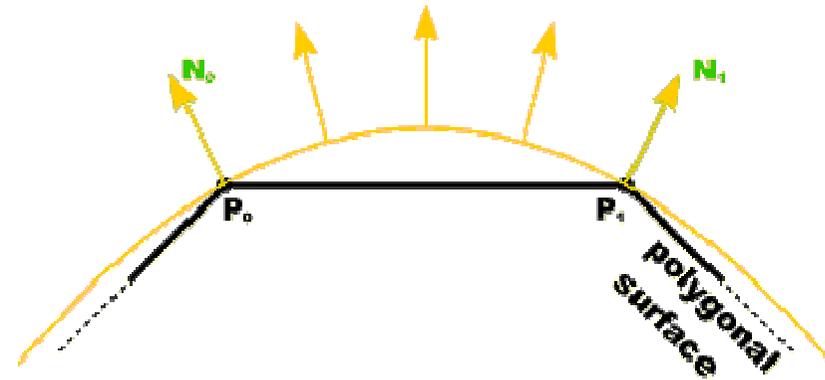
Schattierung

Gouraud



- Interpolation der Intensitäten
- Einfache Berechnung
- Hohe Geschwindigkeit

Phong

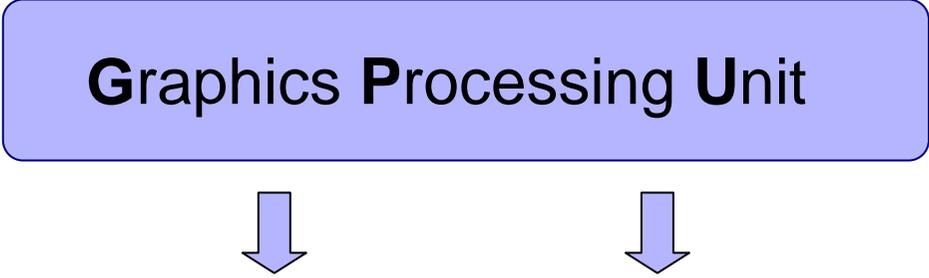


- Interpolation der Normalen
- Normalisierung pro Pixel(!)
- Evaluation des Beleuchtungsmodells pro Pixel(!)
- Hoher Rechenaufwand

GPU

- Bisher:**
- Transformation/Schattierung auf CPU
 - Übergabe der transformierten/schattierten Polygon an die Grafikhardware
- Besser:** Effiziente *Custom-Chips* für Transformation und Schattierung

Graphics Processing Unit



- Entlastung der CPU
- Höhere Effizienz als *general purpose* CPUs

GPU

Eingangsdaten:

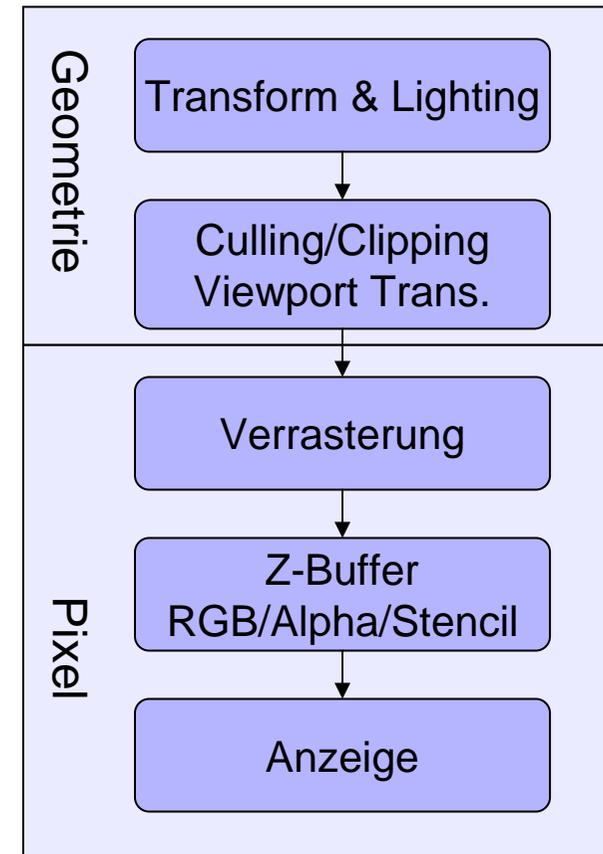
- *Strom* aus Koordinaten / Normalen
- Lichtquellen
- Transformationen

➔ Stream Processing

➔ SIMD Architektur (*Unit*)

➔ Mehrere *Units* parallel (MIMD)

➔ Verstecken von Latenzzeiten durch *Pipelining*



Stream Processing

Stream: (geordneter) Datenstrom von einfachen oder auch komplexen Datentypen

Kernel: Programm, das Berechnungen auf einem Stream ausführt

Eigenschaften:

- Keine Datenabhängigkeiten zwischen den Stream-Elementen
- Eingabe ist *read-only*
- Ausgabe ist *write-only*
- Eingeschränkter Kontrollfluss

Stream Processing

Konsequenzen:

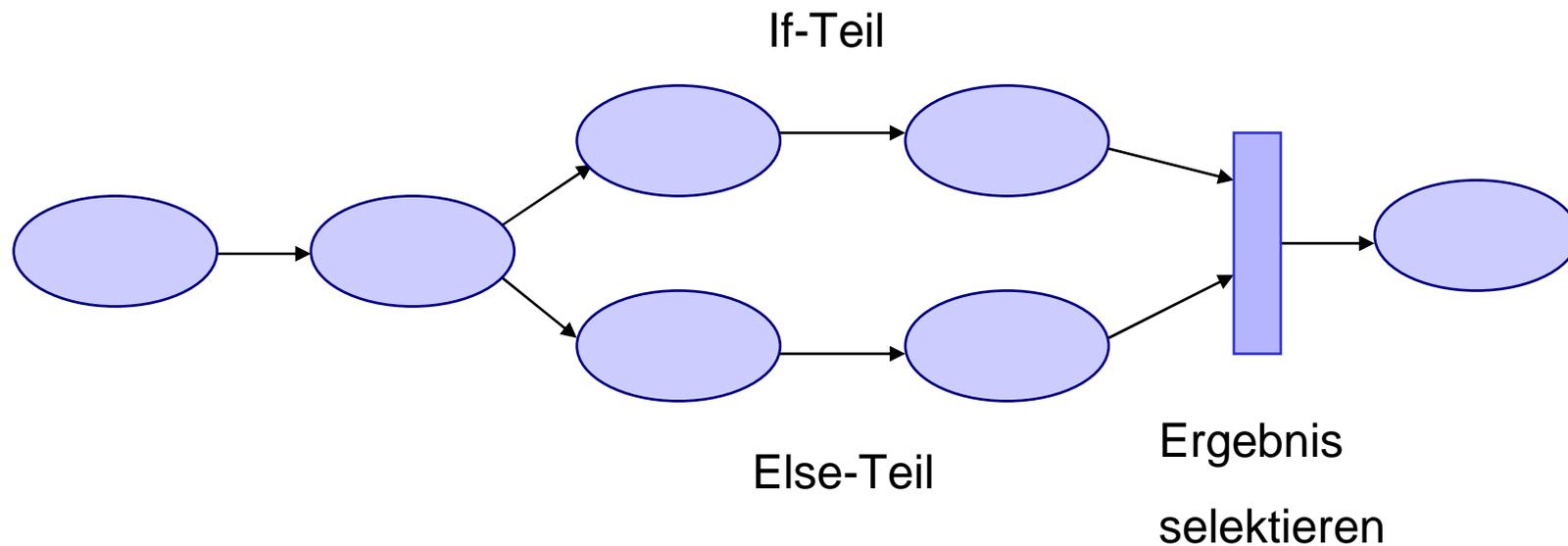
- Hohe Datenparallelität (keine Abhängigkeiten im Stream)
- Tiefe Pipelines (Verketteten von Kernels)
- Hohe Lokalität (Register / Caching)
- **Eingeschränkte Programmierung**

Beispiel: Verzweigung (if-then-else Anweisungen)

- grundsätzlich vermeiden
- in einigen Architekturen nicht ohne weiteres möglich

Stream Processing

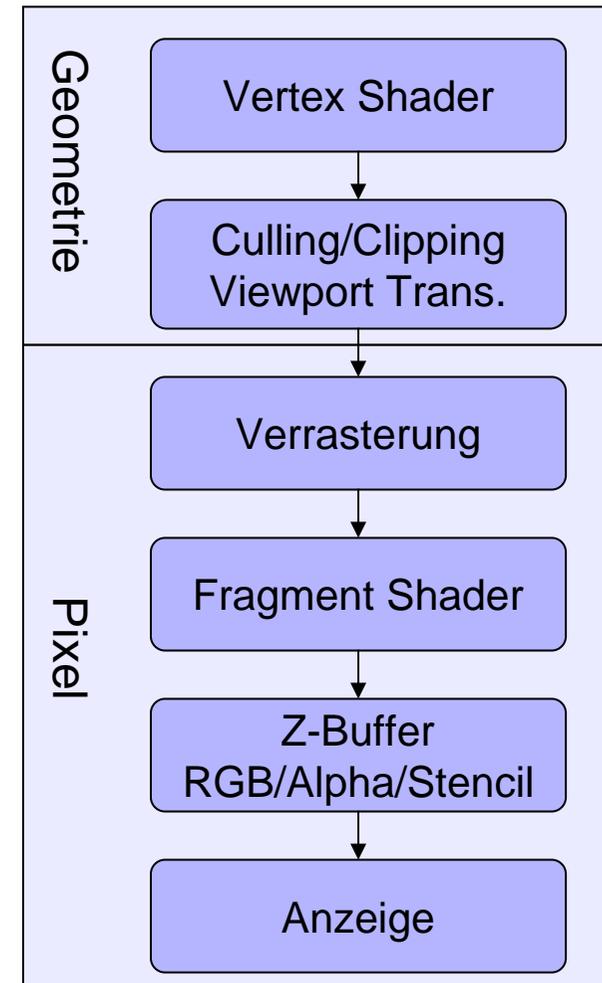
Möglicher Workaround: if **und** else Teil ausführen



- Moderne GPUs unterstützen *dynamic branching*

Programmierbare Shader

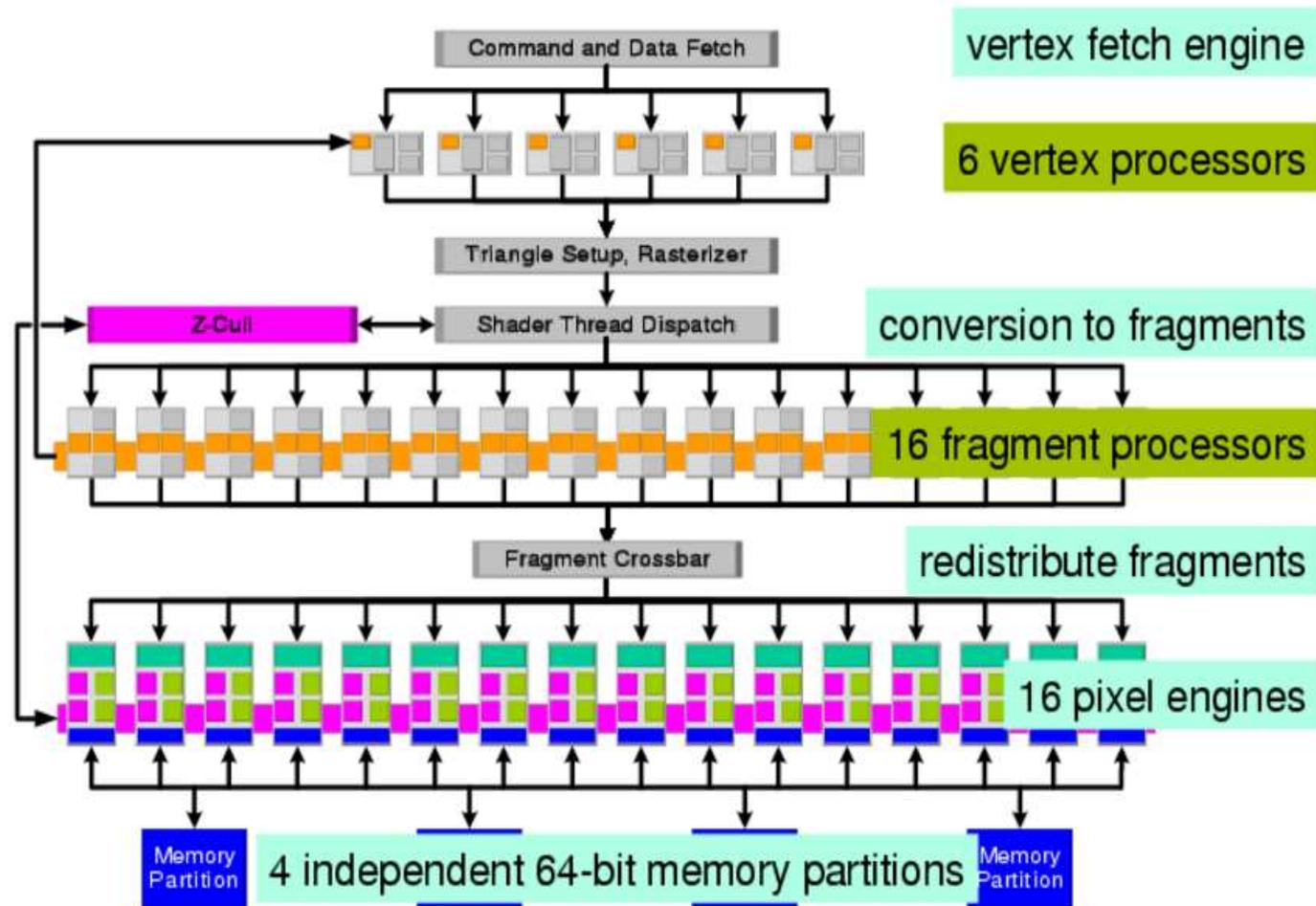
- Erhöhung der Flexibilität
- Verfeinerung der Schattierung
 - Komplexe Beleuchtungsmodelle (z.B. Anisotropie)
 - BRDFs
- Verbesserung der Texturierung
 - Texturverarbeitung / Bumpmapping / prozedurale Texturen
- Effekte/Animationen in Echtzeit



nVidia GeForce 6800

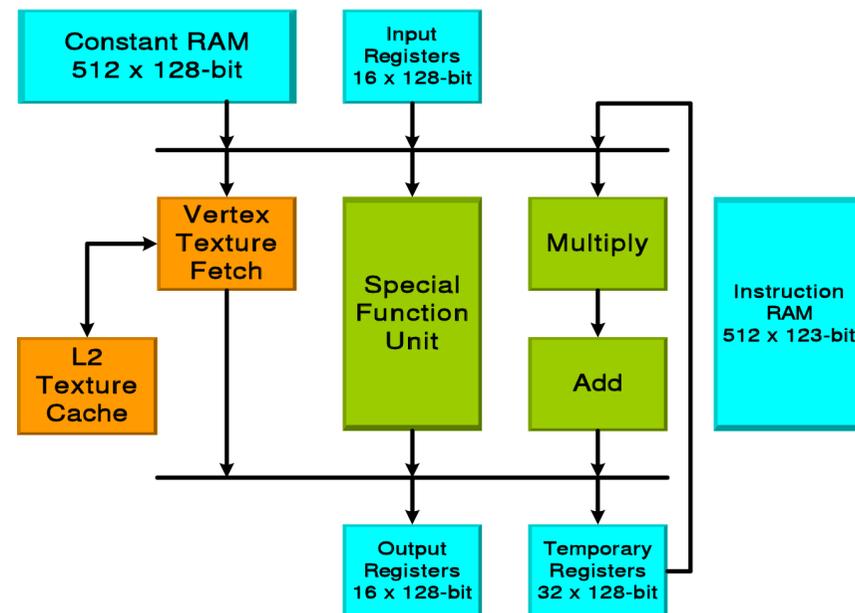
- 222 Mio. Transistoren (P4 Extreme Ed.: 188 Mio.)
- 400+ MHz GPU-Takt (üblich: ca. 425 MHz)
- 550 MHz RAM (256-Bit Bus, PC dual-channel: 128-Bit)
- ~ 120 GFLOP/s peak performance
- ~ 35GB/s memory bandwidth
- 6 Vertex Units
- 16 Fragment Units

nVidia GeForce 6800



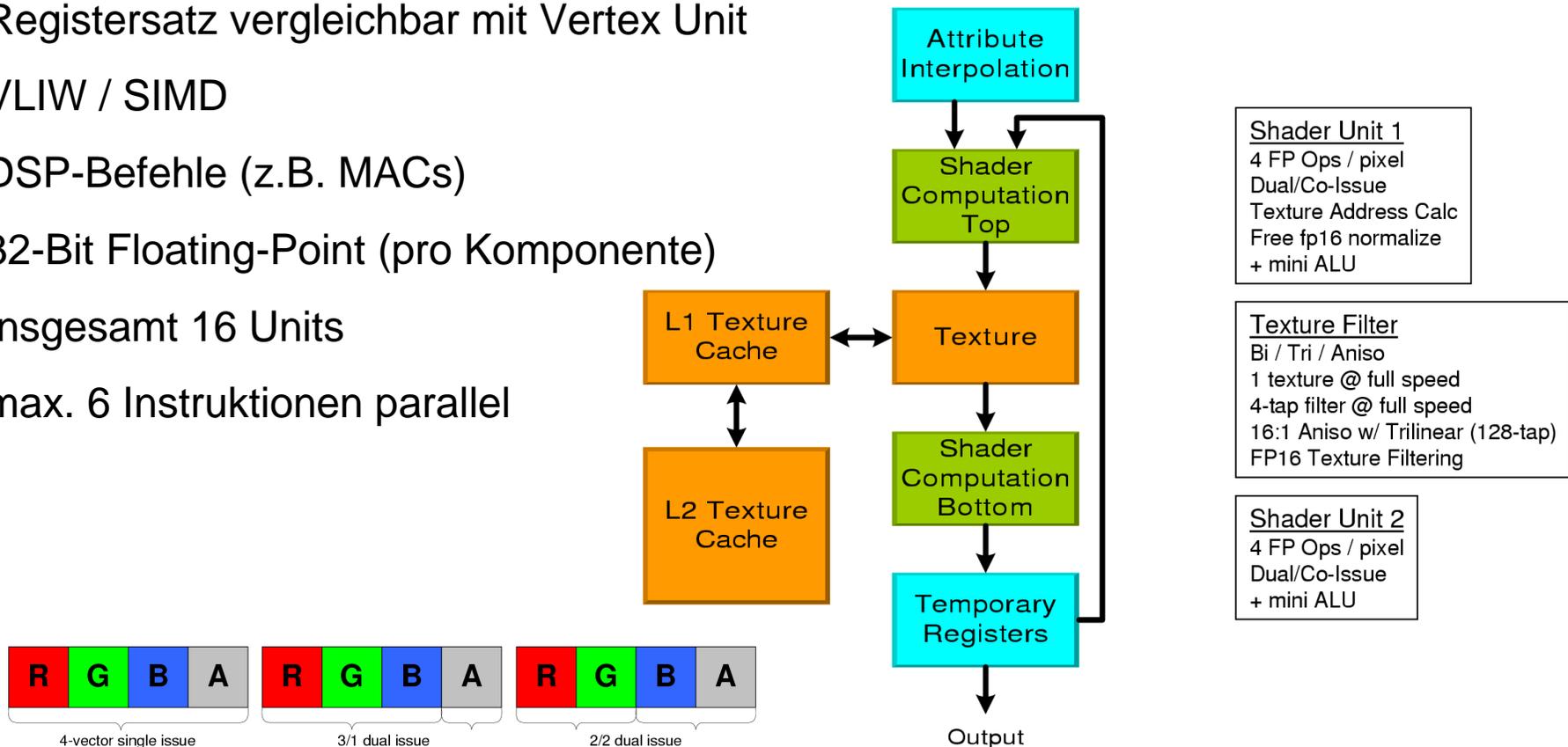
GeForce 6800 Vertex Unit

- 6 Vektor Fließkommprozessoren (6 Vertex Shader Units)
 - VLIW Architektur
 - 512 x 128-Bit Context RAM
 - 32 x 128-Bit Register
 - 16 x 128-Bit In/Out Register
 - Platz für 512 Instruktionen
 - 32-Bit Floating-Point
- Bis 3 Threads pro Vertex Unit
 - Verstecken der Latenz



GeForce 6800 Fragment Unit

- Registersatz vergleichbar mit Vertex Unit
- VLIW / SIMD
- DSP-Befehle (z.B. MACs)
- 32-Bit Floating-Point (pro Komponente)
- Insgesamt 16 Units
- max. 6 Instruktionen parallel



GeForce 6800 Fragment Unit

```

ps_2_0
def c1, 2.0, -1.0, 0.0, 0.0
dcl t0.rg
dcl t1
dcl t4.rgb
dcl v0
dcl_2d s0
dcl_2d s1
dcl_cube s2
dcl_2d s3

# clock 1
texld r0, t0, s0;      # tex fetch
madr r0,r0,c1.r,c1.g  # tex
nrm r1.rgb, t4        # shdr0
dp3 r1.r,r1,r0        # shdr1
mul r0.a,r0,r0        # shdr1

# clock 2
mul r1.a,r0.a,c2.a    # shdr0
mul r0.rgb,r1.r,r0    # shdr0
add r0.a,r1.r,r1.r    # shdr0
mad r0.rg,r0.a,c1,c1.a # shdr1
mul r1.ba,r1.a,r0.a,c2 # shdr1

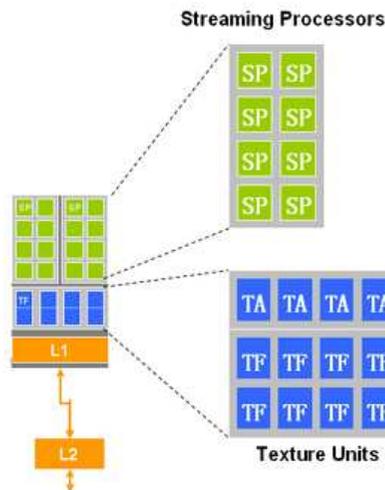
# clock 3
rcp r0.a,r0.a         # shdr0
mul r0.rgr0,r0.a     # shdr0
mul r0.a,r0.a,r1.a    # in shdr0
texld r2,r0, s1      # tex fetch
mad r2.rgb,r0.a,r2,c5 # shdr1
abs r0.a,r0.a         # shdr1
log r0.a,r0.a         # shdr1
<< etc >>
mov oC0, r0          # output
color

```

nVidia GeForce 8800

- 680 Mio. Transistoren (P4 Extreme Ed.: 188 Mio.)
- 1.35 GHz GPU-Takt
- 575 MHz RAM-Takt (384-Bit Bus, PC dual-channel: 128-Bit)
- ~ 520 GFLOP/s peak performance
- ~ 86 GB/s memory bandwidth
- 8 Gruppen von je 16 Streamprozessoren

GeForce 8800



- **SP = Streaming Processors**
- **TF = Texture Filtering Unit**
- **TA = Texture Address Unit**
- **L1/L2 = Caches**

- Skalare Prozessoren
- Dual-issue MAD/MUL
- Gruppierung zu Einheiten
- Verteilung auf Workload

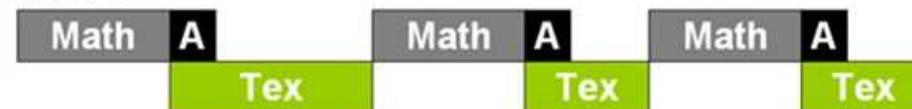
- Umstieg auf skalare Prozessoren bedingt durch Auswertung von Shaderprogrammen
- Bessere Auslastung der Shader-Einheiten
- 2-fache Geschwindigkeitssteigerung (lt. nVidia)

GeForce 8800

Weitere Features:

- Extremes Multithreading (1000+ Threads)
 - Verstecken der Texture-Fetch Latenzen
- Verbesserung des Branchings (Batches kleiner)
- Geometry-Shader hinzugefügt

- G7x :



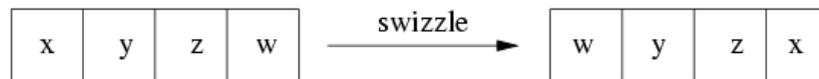
- G8x :



Shaderprogrammierung

- Vektorarithmetik (4-Komponenten Vektor, vgl. Intel SSE)
- Horizontale und Vertikale Arithmetik
 - Skalarprodukt
 - Vektoraddition
 - Matrix/Vektor Multiplikation
 - ...

- Swizzling:



- Writemask

GL Shading Language (GLSL)

- Hochsprache für Shader-Programmierung
- Syntax sehr C-ähnlich
- Vertex- und Fragmentprogramme
- Source wird zur Laufzeit kompiliert und gelinked
- Parameterübergabe vom Vertex ins Fragmentprogramm möglich

GLSL Beispiel

- Einfaches Beispiel für Volumenvisualisierung (Farb-LUT):

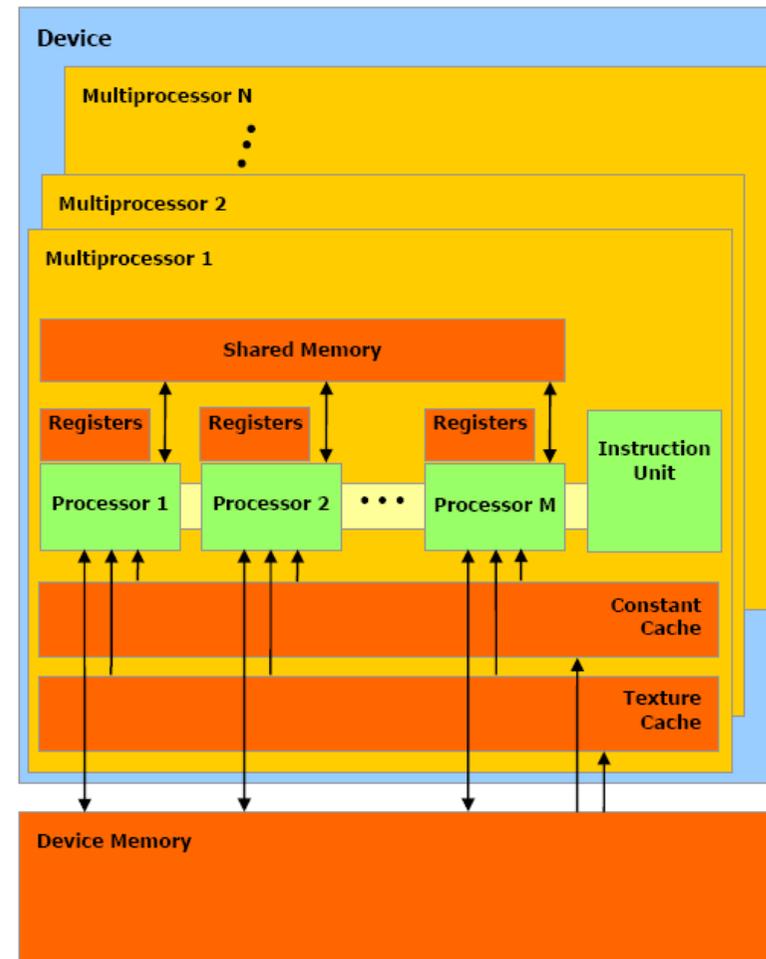
```
uniform sampler3D VolumeTexture;  
uniform sampler2D TransferTable;  
  
void main() {  
    float voxel;  
    vec2 transferindex;  
    vec4 ambient;  
    voxel=texture3D(VolumeTexture,gl_TexCoord[0].xyz).r;  
    transferindex=vec2(voxel,0.0);  
    ambient=texture2D(TransferTable,transferindex);  
    gl_FragColor=ambient;  
}
```

CUDA

- Eingeführt mit der GeForce 8800
- Framework für allg. Stream-Programming (nicht Grafik-spezifisch)
- Arbeitet mit hohen Zahlen an Threads (latency hiding)
- Syntax als Erweiterung von C

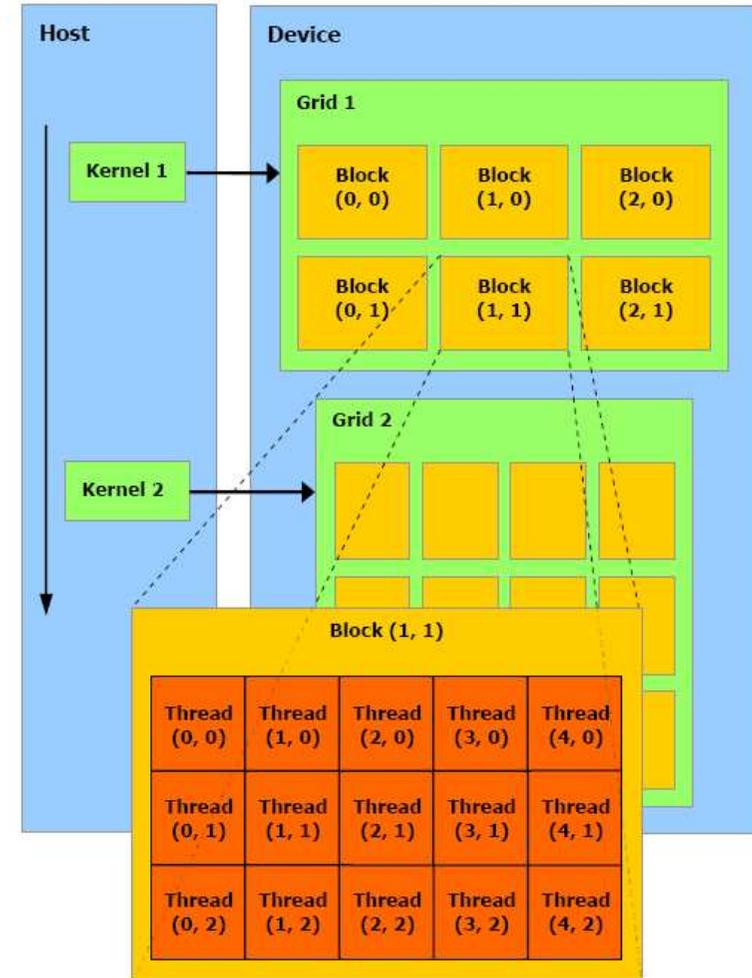
CUDA

- Zusammenfassen der Prozessoren zu *Multiprozessoren*
- Gemeinsame Caches für jeweils einen Multiprozessor
- Alle Einzelprozessoren führen denselben Befehl aus (SIMD)



CUDA

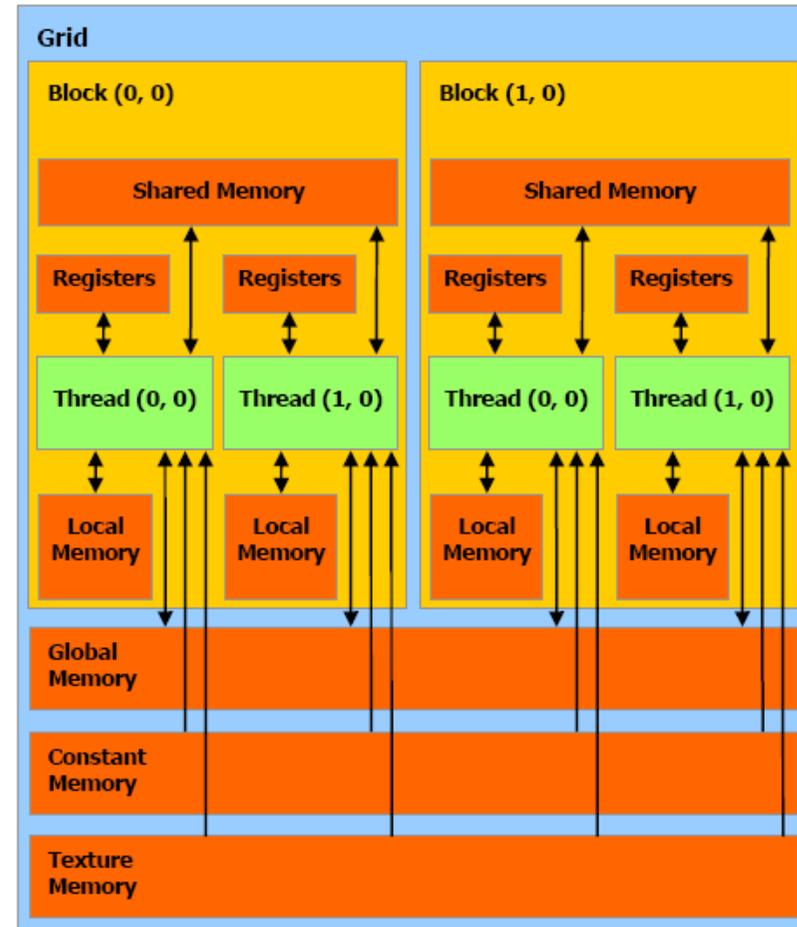
- Einteilung in Thread-Blocks
- Threads in einem Block teilen sich Daten
- Kommunikation/Synchronisation nur innerhalb desselben Blockes
- Blöcke werden zu *Batches* zusammengefasst



CUDA

Scheduler führt Threads in Blöcken
(warps) aus (32 Threads)

- Dynamic register allocation
- Reihenfolge der *warp* Ausführung nicht definiert, Synchronisation programmatisch möglich
- 8 Prozessoren = 1 Multiprozessor
- 16 KiB Shared Memory
- 8192 Register pro Multiprozessor
- Lokaler Speicher als Teil des Framebuffers modelliert
- Shared Memory ist *on-chip*



CUDA

Performance (GeForce 8800GTX):

- Multiply bzw. Multiply-Add: 4 Takte pro warp
- RCP/RCPSQRT/Log: 16 Takte pro warp
- FP Division: 20-36 Takte pro warp

Peak Performance: $2 \cdot (32/4) \cdot 16 \cdot 1.35 = 345.6 \text{ GFLOP/s}$

BLAS3 Operationen (Matrix/Matrix Multiplikation):

- GF8800: 100+ GFLOP/s
- Cell BE: 175 GFLOP/s

CUDA (Beispiel)

```
__global__ void cudaTest(float *array, int arraySize) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < arraySize) array[idx] = (array[idx] * array[idx]) / 4.0f;
}

int main(void) {
    float *cpuArray, deviceArray;
    const int arraySize = 10;
    cpuArray = (float *)malloc(arraySize * sizeof(float));
    cudaMalloc((void **)&deviceArray, arraySize * sizeof(float));
    for (int i=0; i < arraySize; i++) cpuArray[i] = (float)i;
    cudaMemcpy(deviceArray, cpuArray, arraySize * sizeof(float), cudaMemcpyHostToDevice);

    int blockSize = 4;
    int numBlocks = arraySize / blockSize + (arraySize % blockSize == 0 ? 0 : 1);
    cudaTest <<< numBlocks, blockSize >>> (deviceArray, arraySize);
    cudaMemcpy(cpuArray, deviceArray, arraySize * sizeof(float), cudaMemcpyDeviceToHost);

    free(cpuArray);
    cudaFree(deviceArray);
}
```

Entwicklung

- Ausbau der Parallelität (mehr Units)
- Erhöhung der RAM-Geschwindigkeit
- Benutzung der GPU nicht nur für Grafik
- Derzeitige Entwicklung der Grafikhardware (noch) schneller als Entwicklung der CPUs

Weitere Informationen

- Vorlesung „Mensch-Maschine Interaktion“

<http://www.opengl.org>

<http://developer.nvidia.com>

<http://www.ati.com>

<http://www.gpgpu.org>